Data-Flow Analysis

Basic Blocks

One entrance point (first instruction).

- New BB for each labeled statement.

One exit point (last instruction).

- New BB after each `jump`/`return`.
- No new BB after `call`.

```
lt_true:                ;; less than
                        push fp
                        push r0
                        la r2 <- Bool..new
                        call r2
                        pop r0
                        pop fp
                        li r2 <- 1
                        st r1[3] <- r2
                        jmp lt_end
lt_bool:                ;; two Bools
lt_int:                 ;; two Ints
                        ld r1 <- fp[3]
                        ld r2 <- fp[2]
                        ld r1 <- r1[3]
                        ld r2 <- r2[3]
                        blt r1 r2 lt_true
                        jmp lt_false
lt_string:              ;; two Strings
                        ld r1 <- fp[3]
                        ld r2 <- fp[2]
                        ld r1 <- r1[3]
                        ld r2 <- r2[3]
                        ld r1 <- r1[0]
                        ld r2 <- r2[0]
                        blt r1 r2 lt_true
                        jmp lt_false
lt_end:                 pop ra
                        li r2 <- 2
                        add sp <- sp r2
```

Basic Blocks

One entrance point (first instruction).

- New BB for each labeled statement.

One exit point (last instruction).

- New BB after each `jump`/`return`.
- No new BB after `call`.

```
lt_true:                ;; less than
                        push fp
                        push r0
                        la r2 <- Bool..new
                        call r2
                        pop r0
                        pop fp
                        li r2 <- 1
                        st r1[3] <- r2
                        jmp lt_end
-------------------------------------------------
lt_bool:                ;; two Bools
lt_int:                 ;; two Ints
                        ld r1 <- fp[3]
                        ld r2 <- fp[2]
                        ld r1 <- r1[3]
                        ld r2 <- r2[3]
                        blt r1 r2 lt_true
-------------------------------------------------
                        jmp lt_false
-------------------------------------------------
lt_string:              ;; two Strings
                        ld r1 <- fp[3]
                        ld r2 <- fp[2]
                        ld r1 <- r1[3]
                        ld r2 <- r2[3]
                        ld r1 <- r1[0]
                        ld r2 <- r2[0]
                        blt r1 r2 lt_true
-------------------------------------------------
                        jmp lt_false
-------------------------------------------------
lt_end:                 pop ra
                        li r2 <- 2
                        add sp <- sp r2
```

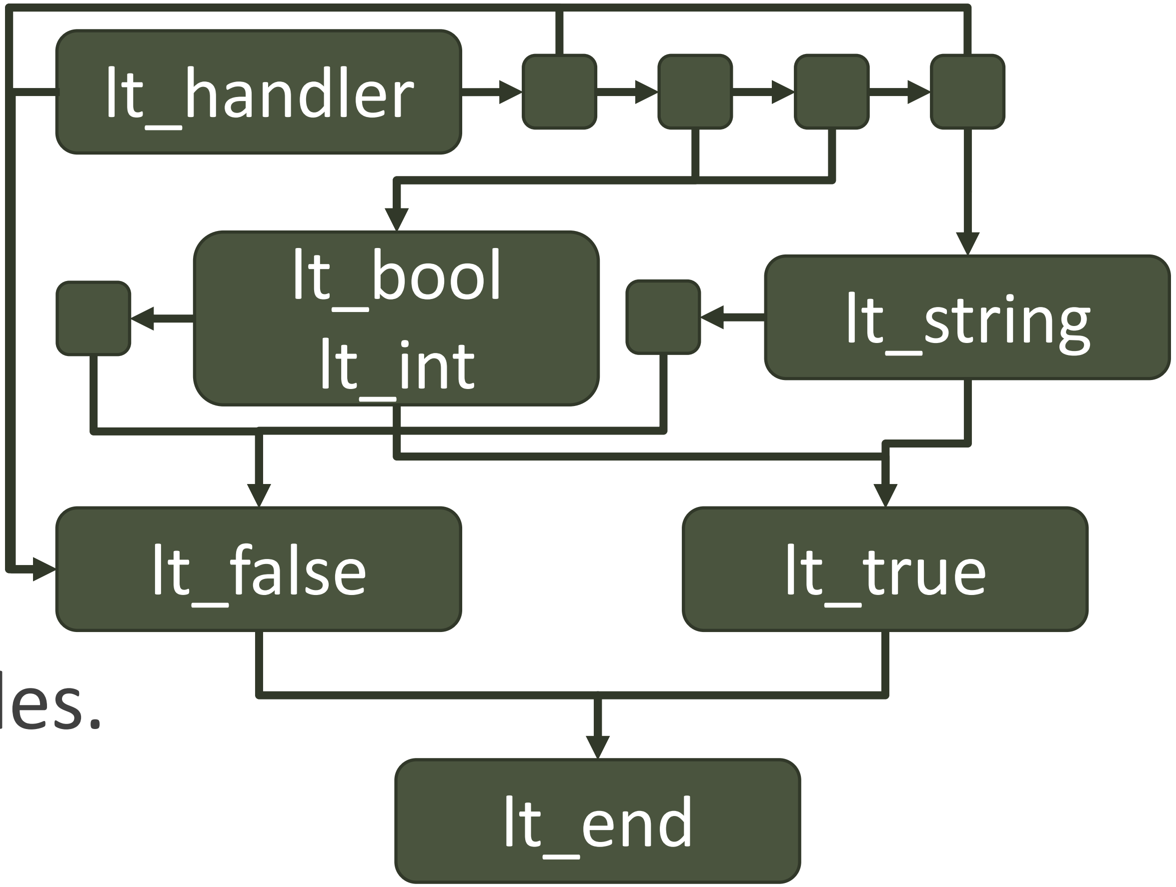
Control Flow Graphs (CFGs)

Directed, potentially cyclic.

- Nodes: basic blocks.
- Edges: jumps, fall-through.

One graph per function.

Unique entrance and exit nodes.



Example: Always Null Dereference

```
let x : Object in
  x.type_name()
```

Example: Always Null Dereference

```
let x : Object in
  x.type_name()
```

```
let x : Object in
  if (isnull x)
    nullDerefError
  else
    x.type_name()
  fi
```

Example: Always Null Dereference

```
let x : Object in
  if (isnull x)
    nullDerefError
  else
    x.type_name()
  fi
```

x is always null here. We do not need to check.

nullDerefError

How can our compiler recognize this and simplify?

Example: Always Null Dereference

```
let x : Object in
  if (isnull x)
    nullDerefError
  else
    x.type_name()
  fi
```

What does this compile to?

Example: Always Null Dereference

```
let x : Object in
  if (isnull x)
    nullDerefError
  else
    x.type_name()
  fi
```

```
 li r1 <- 0        ; let x : Object
 li r0 <- 0        ; if (isnull x)
 beq r0 r1 L_if_true
 push r1           ; pass self arg
 ld r1 <- r1[2]    ; get vtable
 ld r1 <- r1[3]    ; get type_name
 call r1
 jmp L_if_end
L_if_true:
 li r1 <- 2        ; error on line 2
 call null_deref_error
L_if_end:
```

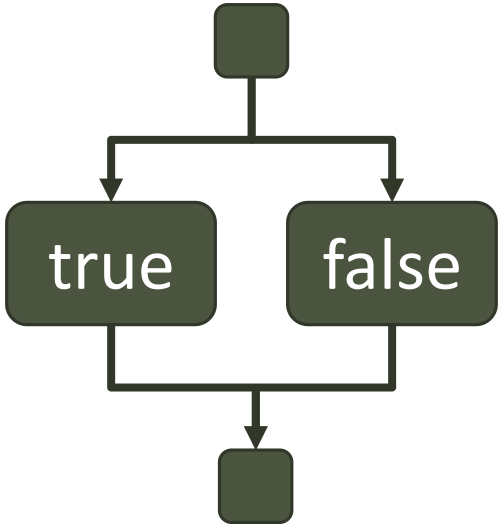
Example: Always Null Dereference

```
let x : Object in
  if (isnull x)
    nullDerefError
  else
    x.type_name()
  fi
```

```
 li r1 <- 0          ; let x : Object
 li r0 <- 0          ; if (isnull x)
 beq r0 r1 L_if_true
---------------------------------------
 push r1             ; pass self arg
 ld r1 <- r1[2]      ; get vtable
 ld r1 <- r1[3]      ; get type_name
 call r1
 jmp L_if_end
---------------------------------------
L_if_true:
 li r1 <- 2          ; error on line 2
 call null_deref_error
---------------------------------------
L_if_end:
```

Example: Always Null Dereference

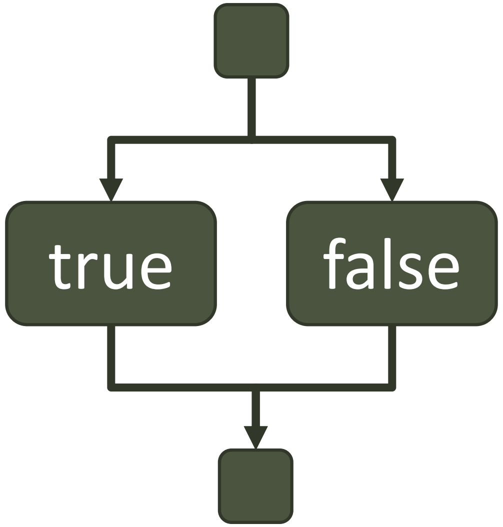
```
 li r1 <- 0
 li r0 <- 0
 beq r0 r1 L_if_true
-----------------------------------
 push r1
 ld r1 <- r1[2]
 ld r1 <- r1[3]
 call r1
 jmp L_if_end
-----------------------------------
L_if_true:
 li r1 <- 2
 call null_deref_error
-----------------------------------
L_if_end:
```



Example: Always Null Dereference

```
 li r1 <- 0
 li r0 <- 0
 beq r0 r1 L_if_true
-----------------------------------
 push r1
 ld r1 <- r1[2]
 ld r1 <- r1[3]
 call r1
 jmp L_if_end
-----------------------------------
L_if_true:
 li r1 <- 2
 call null_deref_error
-----------------------------------
L_if_end:
```

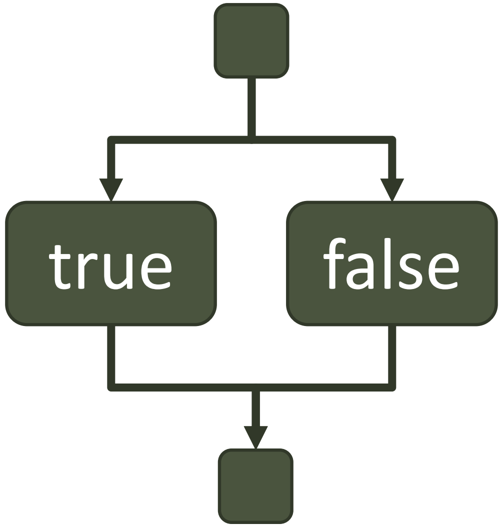
{r0: ?, r1: 0}



Example: Always Null Dereference

```
  li r1 <- 0
  li r0 <- 0
  beq r0 r1 L_if_true
-----------------------------------
  push r1
  ld r1 <- r1[2]
  ld r1 <- r1[3]
  call r1
  jmp L_if_end
-----------------------------------
L_if_true:
  li r1 <- 2
  call null_deref_error
-----------------------------------
L_if_end:
```

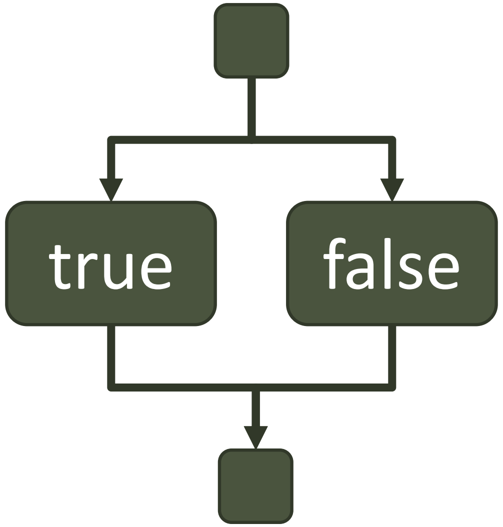
{r0: 0, r1: 0}



Example: Always Null Dereference

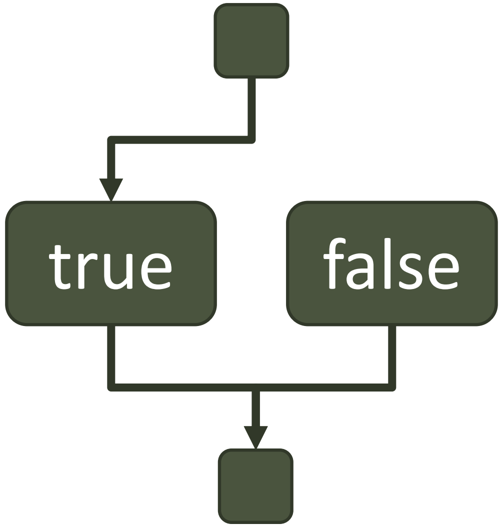
```
 li r1 <- 0
 li r0 <- 0
 beq r0 r1 L_if_true
-----------------------------------
 push r1
 ld r1 <- r1[2]
 ld r1 <- r1[3]
 call r1
 jmp L_if_end
-----------------------------------
L_if_true:
 li r1 <- 2
 call null_deref_error
-----------------------------------
L_if_end:
```

{r0: 0, r1: 0} replace with `jmp L_if_true`



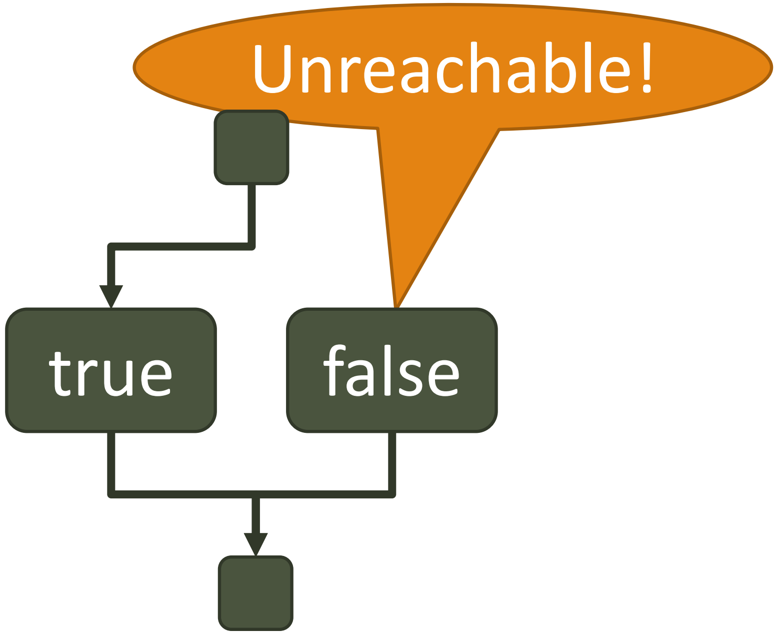
Example: Always Null Dereference

```
  li r1 <- 0
  li r0 <- 0
  jmp L_if_true
-----------------------------------
  push r1
  ld r1 <- r1[2]
  ld r1 <- r1[3]
  call r1
  jmp L_if_end
-----------------------------------
L_if_true:
  li r1 <- 2
  call null_deref_error
-----------------------------------
L_if_end:
```



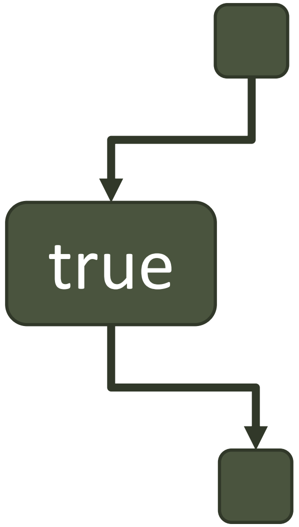
Example: Always Null Dereference

```
  li r1 <- 0
  li r0 <- 0
  jmp L_if_true
-----------------------------------
  push r1
  ld r1 <- r1[2]
  ld r1 <- r1[3]
  call r1
  jmp L_if_end
-----------------------------------
L_if_true:
  li r1 <- 2
  call null_deref_error
-----------------------------------
L_if_end:
```



Example: Always Null Dereference

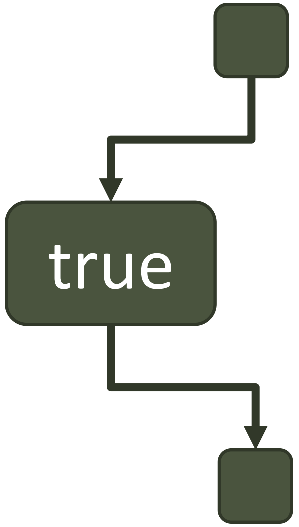
```
  li r1 <- 0
  li r0 <- 0
  jmp L_if_true
-----------------------------------
L_if_true:
  li r1 <- 2
  call null_deref_error
-----------------------------------
L_if_end:
```



Example: Always Null Dereference

```
  li r1 <- 0
  li r0 <- 0
  jmp L_if_true
L_if_true:
  li r1 <- 2
  call null_deref_error
L_if_end:
```

Peephole: remove jmp



Example: Always Null Dereference

```
  li r1 <- 0
  li r0 <- 0
  li r1 <- 2
  call null_deref_error
-----------------------------------
L_if_end:
```

Can we eliminate these?

Data-Flow Analysis

Considers transformations along all possible paths.

- What is wrong with this goal?

Data-Flow Analysis

Considers transformations along all possible paths.

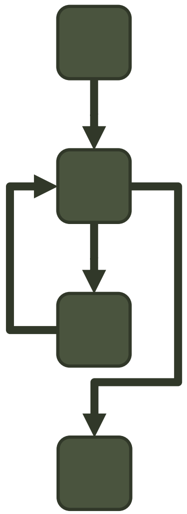
- ***Undecidable*** (solves the Halting Problem).
- Use ***conservative approximation*** instead.

Path approximation:

- Inside basic block, one path from top to bottom.
- Between basic blocks, one path along each edge.

Search Space Explosion

A CFG with ∞ paths!



Needs more approximation.

- Merge data reaching a node along multiple paths.
- Abstract program state to simpler ***data-flow value***.

Data-Flow Analysis

Goal: Determine data-flow value before and after every statement (or basic block).

- Denoted $\textsc{in}[s]$ and $\textsc{out}[s]$ respectively.

Transfer functions: $\textsc{out}[s] = f_s(\textsc{in}[s])$

Meet operator: $\textsc{in}[s] = \bigwedge_{t \text{ precedes } s} \textsc{out}[t]$

Transfer Functions

Property	Definition
Identity Function	$\exists I \in F. \forall x \in V. I(x) = x$
Closed under Composition	$\forall f, g \in F. h(x) = g\big(f(x)\big) \Rightarrow h \in F$

Monotone (1)	$\forall x, y \in V. \forall f \in F$ $f(x \wedge y) \leq f(x) \wedge f(y)$
Monotone (2)	$\forall x, y \in V. \forall f \in F$ $x \leq y \Rightarrow f(x) \leq f(y)$

Data-Flow Analysis

Data-Flow Analysis Framework $(\mathrm{D}, \mathrm{V}, \wedge, F)$

- D: direction (forwards or backwards).
- V: domain of values
- $\wedge$: meet operator
- F: family of transfer functions $V \rightarrow V$

V and $\wedge$ form a ***meet-semilattice*** (***lower semilattice***).

Meet-Semilattices

Property	Example
Idempotent	$x \wedge x = x$
Commutative	$x \wedge y = y \wedge x$
Associative	$x \wedge (y \wedge z) = (x \wedge y) \wedge z$
Partially Ordered	$x \leq y \Leftrightarrow x \wedge y = x$
Top (⊤)	$\forall x. \top \wedge x = x$
Bottom (⊥)	$\forall x. \bot \wedge x = \bot$

Meet-Semilattices

Property	Example
Idempotent	$x \wedge x = x$
Commutative	$x \wedge y = y \wedge x$
Associative	$y) \wedge z$
Partially Ordered	$x \leq y \Leftrightarrow x \wedge y = x$
Top ($\top$)	$\forall x. \top \wedge x = x$
Bottom ($\bot$)	$\forall x. \bot \wedge x = \bot$

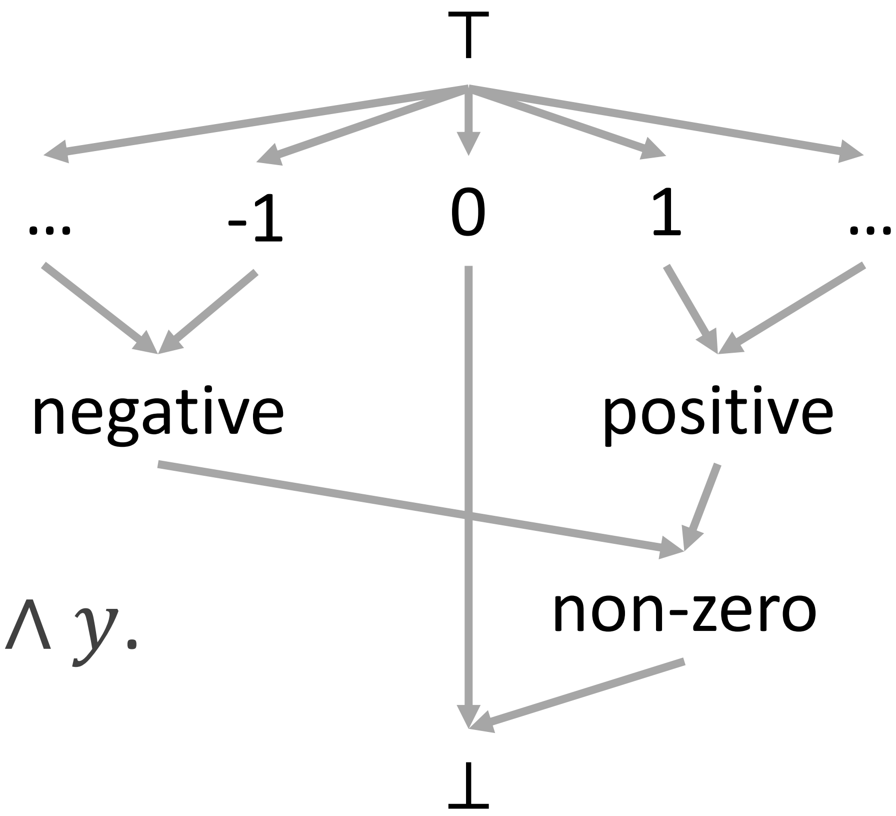
Imply finite height.

Greatest Lower Bound (glb)

g is a glb w.r.t. x and y iff:

- $g \leq x$
- $g \leq y$
- $\forall z. z \leq x \text{ and } z \leq y \Rightarrow z \leq g$

The unique glb of x and y is $g = x \wedge y$.



Data-Flow Algorithm

Given

- $(D, V, \wedge, F)$
- CFG with labeled ENTRY and EXIT nodes.
- v_{ENTRY}

1. For each block B, $\text{OUT}[B] = \top$
2. $\text{OUT}[\text{ENTRY}] = v_{\text{ENTRY}}$
3. While any OUT changes
   1. For each block B except ENTRY
      1. $\text{IN}[B] = \bigwedge_P \text{OUT}[P]$
      2. $\text{OUT}[B] = f_B(\text{OUT}[B])$

Never Null Dereference

```
let x : Object in {
  if (y < 10) then
    x <- “hello”
  else
    x <- 2
  fi;
  x.type_name();
}
```

x is never null here. We do not need to check.

Null Dereference Data-Flow Framework

Direction: Forward

Values:

- $\forall i. ri, sp[i] \in \{\top$, null, not-null, $\bot\}$

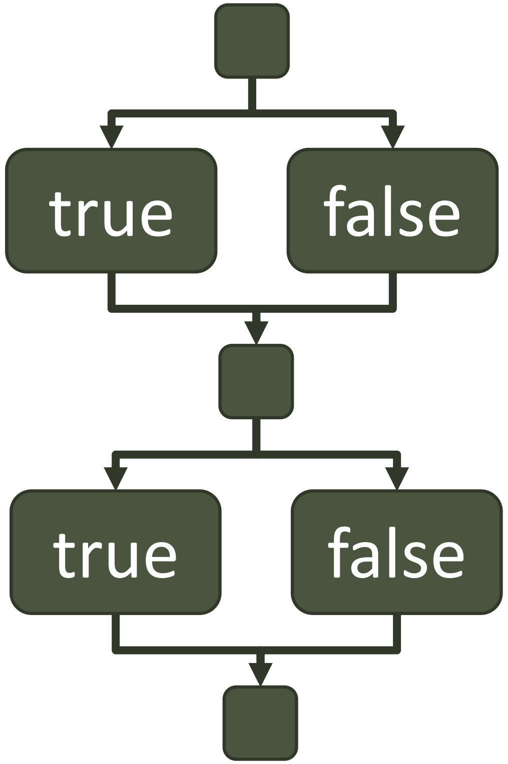
Meet:

- $\forall x, i. ri{:}\top \wedge ri{:}x = ri{:}x$
- $\forall x, i. ri{:}\bot \wedge ri{:}x = ri{:}\bot$
- $\forall i. ri$:null $\wedge$ ri:not-null = $ri{:}\bot$

Statement	Value
`li ri <- 0`	r*i*: null
`li ri <- N`	r*i*: not-null
`la ri <- label`	r*i*: not-null
`st sp[X] <- ri`	sp[*X*]: v(r*i*)
`ld ri <- sp[X]`	r*i*: sp[*X*]
`call X..new`	r1: not-null

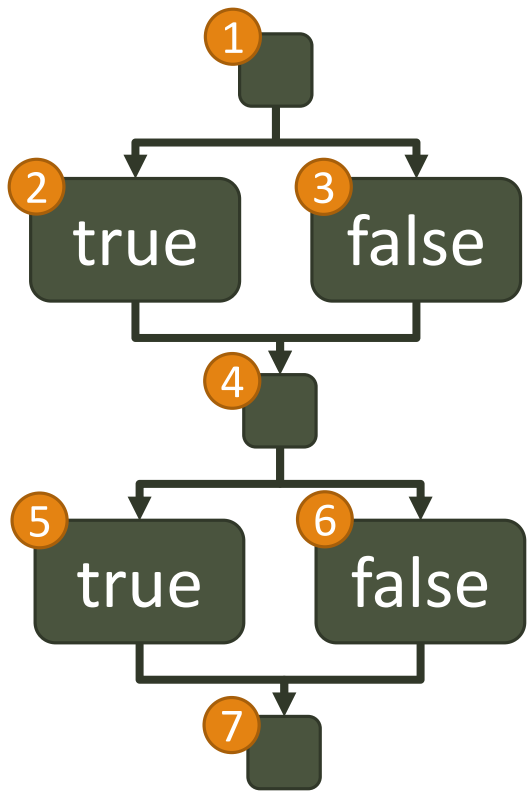
Data-Flow for Null Dereference

```
let x : Object in {
  if (y < 10) then
    x <- “hello”
  else
    x <- 2
  fi;
  x.type_name();
}
```



Data-Flow for Null Dereference

Node	OUT(*node*)
1 (ENTRY)	$r0$: T, $r1$: T, x: T, y: T
2	$r0$: T, $r1$: T, x: T, y: T
3	$r0$: T, $r1$: T, x: T, y: T
4	$r0$: T, $r1$: T, x: T, y: T
5	$r0$: T, $r1$: T, x: T, y: T
6	$r0$: T, $r1$: T, x: T, y: T
7	$r0$: T, $r1$: T, x: T, y: T



Data-Flow Analysis

Use CFG to determine:

- Common sub-expressions for elimination
- Live variables for register allocation / dead code elimination.
- Reaching definitions (constant propagation).
- Loop-invariant computations to lift.
- Induction variables to reduce in strength.

Example: Dead Code Elimination

Direction: Backward

Values:

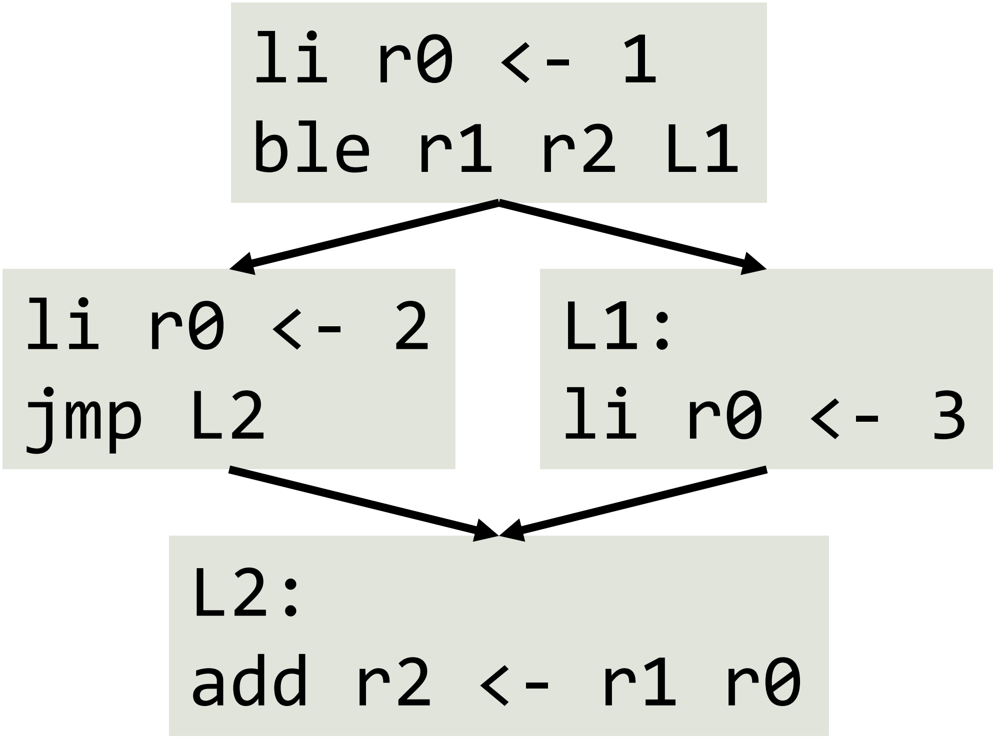
- $\forall i. ri, sp[i] \in \{\top, \bot\}$

Meet:

- Trivial

Transfer Function:

- If ri is an operand, $ri = \bot$



Example: Always Null Dereference

```
  li r1 <- 0
  li r0 <- 0
  li r1 <- 2
  call null_deref_error
-------------------------------
L_if_end:
```

Can we eliminate these?